Find the index of the smallest element in a JavaScript array

devblogs.microsoft.com/oldnewthing/20140526-00

May 26, 2014

Raymond Chen

Today's Little Program isn't even a program. It's just a function.

The problem statement is as follows: Given a nonempty JavaScript array of numbers, find the index of the smallest value. (If the smallest value appears more than once, then any such index is acceptable.)

One solution is simply to do the operation manually, simulating how you would perform the operation with paper and pencil: You start by saying that the first element is the winner, and then you go through the rest of the elements. If the next element is smaller than the one you have, you declare that element the new provisional winner.

```
function indexOfSmallest(a) {
 var lowest = 0;
 for (var i = 1; i < a.length; i++) {
  if (a[i] < a[lowest]) lowest = i;
 }
 return lowest;
}
```

Another solution is to use the `reduce` intrinsic to run the loop, so you merely have to provide the business logic of the initial guess and the `if` statement.

```
function indexOfSmallest(a) {
 return a.reduce(function(lowest, next, index) {
                  return next < a[lowest] : index ? lowest; },
                 0);
}
```

A third solution is to use JavaScript intrinsics to find the smallest element and then convert the element to its index.

```
function indexOfSmallest(a) {
 return a.indexOf(Math.min.apply(Math, a));
}
```

Which one is fastest?

Okay, well, first, before you decide which one is fastest, you need to make sure they are all correct. One thing you discover is that the `min/indexOf` technique fails once the array gets really, really large, or at least it does in Internet Explorer and Firefox. (In my case, Internet Explorer and Firefox gave up at around 250,000 and 500,000 elements, respectively.) That's because you start hitting engine limits on the number of parameters you can pass to a single function. Invoking `apply` on an array of 250,000 elements is the equivalent of calling `min` with 250,000 function parameters.

So we'll limit ourselves to arrays of length at most 250,000.

Before I share the results, I want you to guess which algorithm you think will be the fastest and which will be the slowest.

Still waiting.

I expected the manual version to come in last place, because, after all, it's doing everything manually. I expected the reduce version to be slightly faster, because it offloads some of the work into an intrinsic (though the function call overhead may have negated any of that improvement). I expected the min/indexOf version to be fastest because nearly all the work is being done in intrinsics, and the cost of making two passes over the data would be made up by the improved performance of the intrinsics.

Here are the timings of the three versions with arrays of different size, running on random data. I've normalized run times so that the results are independent of CPU speed.

Elements	manual	reduce	min/indexOf
Internet Explorer 9			
100,000	1.000	2.155	2.739
200,000	1.014	2.324	3.099
250,000	1.023	2.200	2.330
Internet Explorer 10			
100,000	1.000	4.057	4.302
200,000	1.028	4.057	4.642
250,000	1.019	4.091	4.068

Relative running time per array element

Are you surprised? I sure was!

Not only did I have it completely backwards, but the margin of victory for the manual version was way beyond what I imagined.

(This shows that the only way to know your program's performance characteristics for sure is to *sit down and measure it*.)

What I think is going on is that the JavaScript optimizer can do a really good job of optimizing the manual code since it is very simple. There are no function calls, the loop body is just one line, it's all right out there in the open. The versions that use intrinsics end up hiding some of the information from the optimizer. (After all, the optimizer cannot predict ahead of time whether somebody has overridden the default implementation of `Array.prototype.reduce` or `Math.prototype.min`, so it cannot blindly inline the calls.) The result is that the manual version can run over twice as fast on IE9 and over four times as fast on IE10.

I got it wrong because I thought of JavaScript too much like an interpreted language. In a purely interpreted language, the overhead of the interpreter is roughly proportional to the number of things you ask it to do, as opposed to how hard it was to do any of those things. It's like a fixed service fee imposed on every transaction, regardless of whether the transaction was for $100 or 50 cents. You therefore try to make one big purchase (call a complex intrinsic) instead of a lot of small ones (read an array element, compare two values, increment a variable, copy one variable to another).

Bonus chatter: I ran the test on Firefox, too, because I happened to have it handy.

Elements	manual	reduce	min/indexOf
Firefox 16			
100,000	1.000	21.598	3.958
200,000	0.848	21.701	2.515
250,000	0.839	21.788	2.090

Relative running time per array element

The same data collected on Firefox 16 (which sounds ridiculously old because Firefox will be on version 523 by the time this article reaches the head of the queue) shows a different profile, although the winner is the same. The manual loop and the min/indexOf get more efficient as the array size increases. This suggests that there is fixed overhead that becomes gradually less significant as you increase the size of the data set.

One thing that jumps out is that the reduce method way underperforms the others. My guess is that setting up the function call (in order to transition between the intrinsic and the script) is very expensive, and that implementors of the JavaScript engines haven't spent any time optimizing this case because `reduce` is not used much in real-world code.

Update: I exaggerated my naïveté to make for a better narrative. As I point out in the preface to my book, my stories may not be completely true, but they are true enough. Of course I know that JavaScript is jitted nowadays, and that changes the calculus. (Also, the hidden array copy.)

Raymond Chen

Follow